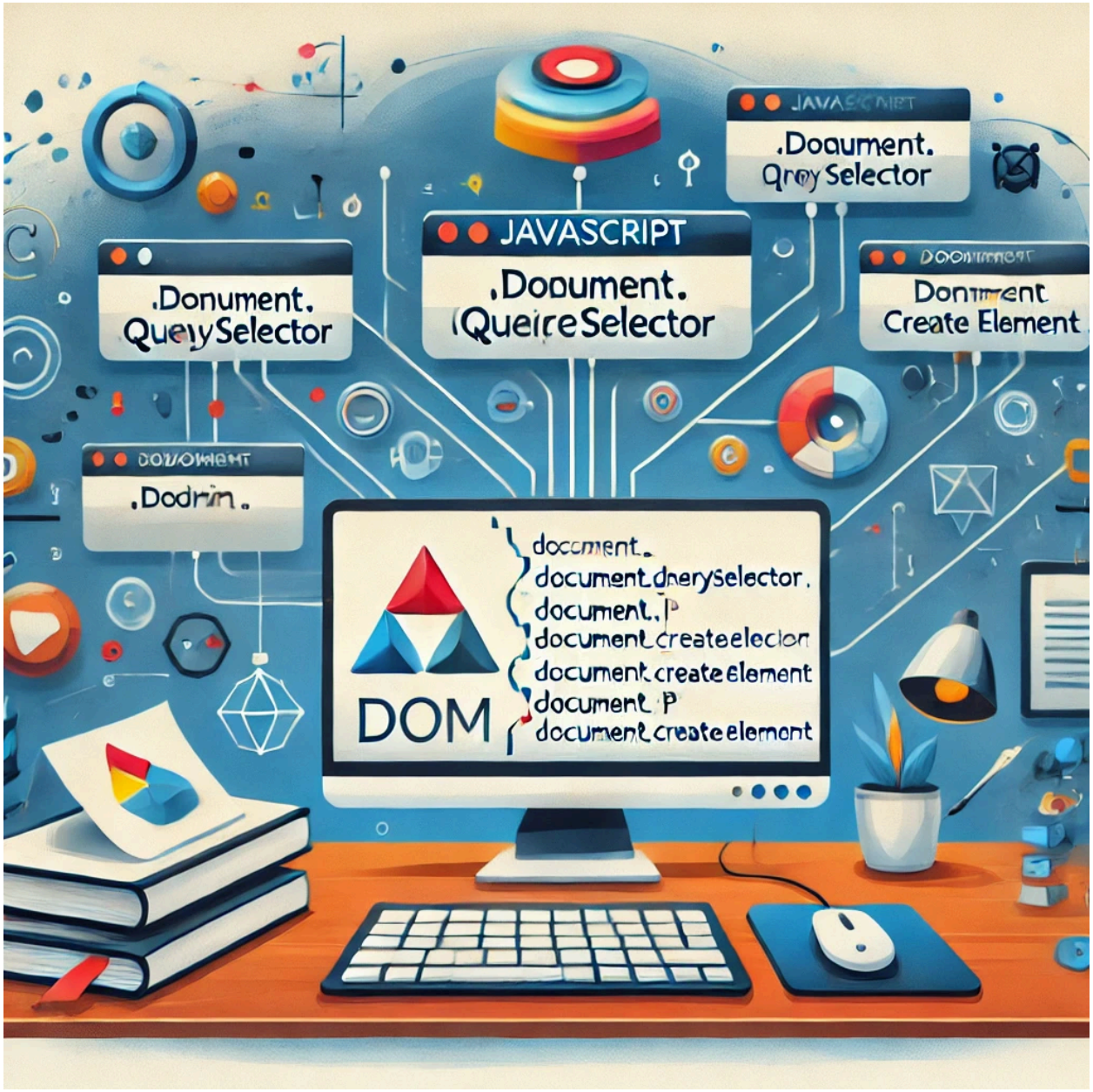
Comprehensive Guide to JavaScript DOM Manipulation



Welcome to the **JavaScript DOM Manipulation** guide! The Document Object Model (DOM) is a crucial concept in web development, allowing developers to interact with and manipulate HTML and XML documents dynamically. This guide will walk you through

the fundamentals of the DOM, how to use JavaScript to interact with it, and provide you with code examples, explanations, exercises, and multiple-choice questions to solidify your understanding.

1. Introduction to the DOM

What is the DOM?

The **Document Object Model (DOM)** is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and

content. The DOM represents the document as a tree of nodes, where each node corresponds to a part of the document (like elements, attributes, and text).

Why Use the DOM?

- **Dynamic Content:** Modify the content of a webpage in response to user interactions without reloading the page.
- **Styling:** Change the appearance of elements dynamically.
- **Interactivity:** Create interactive features like sliders, modals, and form validations.
- **Data Manipulation:** Fetch and display data from external sources seamlessly.

JavaScript and the DOM

JavaScript is the primary language used to interact with the DOM. By using JavaScript, developers can:

- **Select Elements:** Identify and access specific parts of the webpage.
- **Manipulate Elements:** Change content, styles, and attributes.
- **Handle Events:** Respond to user actions like clicks, hovers, and form submissions.
- **Create and Remove Elements:** Dynamically add or remove elements from the page.

2. Selecting Elements

Selecting elements is the first step in DOM manipulation. JavaScript provides various methods to select elements based on different criteria.

Selecting by ID

Each HTML element can have a unique `id` attribute. Selecting elements by ID is one of the most efficient ways.

Syntax:

```
document.getElementById("elementId");
```

Example:

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Selecting by ID</title>
</head>
<body>
    <h1 id="header">Welcome!</h1>
    <script>
        const header = document.getElementById("header");
        console.log(header); // Logs the <h1> element
    </script>
</body>
</html>
```

Selecting by Class Name

Elements can share the same `class` attribute. This method returns a live HTMLCollection of elements.

Syntax:

```
document.getElementsByClassName("className");
```

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Selecting by Class Name</title>
</head>
<body>
    <p class="text">Paragraph 1</p>
    <p class="text">Paragraph 2</p>
    <script>
        const paragraphs =
document.getElementsByClassName("text");
```

```
        console.log(paragraphs); // Logs an HTMLCollection of
<p> elements
    </script>
</body>
</html>
```

Selecting by Tag Name

This method selects all elements of a given tag name, returning a live HTMLCollection.

Syntax:

```
document.getElementsByTagName("tagName");
```

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Selecting by Tag Name</title>
</head>
<body>
    <div>Div 1</div>
    <div>Div 2</div>
    <script>
        const divs = document.getElementsByTagName("div");
        console.log(divs); // Logs an HTMLCollection of <div>
elements
    </script>
</body>
</html>
```

Using `querySelector` and `querySelectorAll`

These modern methods provide powerful ways to select elements using CSS selectors.

- **`querySelector`:** Returns the first matching element.

- **`querySelectorAll`:** Returns a static NodeList of all matching elements.

Syntax:

```
document.querySelector("selector");
document.querySelectorAll("selector");
```

Examples:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Query Selector Example</title>
</head>
<body>
    <div class="container">
        <p class="text">Paragraph 1</p>
        <p class="text highlight">Paragraph 2</p>
        <p>Paragraph 3</p>
    </div>
    <script>
        // Select the first element with class 'text'
        const firstText = document.querySelector(".text");
        console.log(firstText); // Logs the first <p> with class
'text'
        // Select all elements with class 'text'
        const allTexts = document.querySelectorAll(".text");
        console.log(allTexts); // Logs a NodeList of <p>
elements with class 'text'
        // Select the element with class 'highlight' inside
'container'
        const highlight = document.querySelector(".container
.highlight");
        console.log(highlight); // Logs the <p> with classes
'text highlight'
```

```
    </script>
</body>
</html>
```

Advantages of `querySelector` and `querySelectorAll`:

- **Flexibility:** Utilize complex CSS selectors.
- **Consistency:** Works uniformly across different selection criteria.
- **Static NodeList:** `querySelectorAll` returns a static list, avoiding issues with live collections.

3. Manipulating Elements

Once elements are selected, you can manipulate them in various ways—changing their content, styles, attributes, or even creating and removing them.

Changing Content

Using `innerHTML`

`innerHTML` allows you to get or set the HTML content within an element.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Changing Content with innerHTML</title>
</head>
<body>
    <div id="content">
        <p>Original Paragraph</p>
    </div>
    <button id="changeBtn">Change Content</button>
    <script>
        const changeBtn = document.getElementById("changeBtn");
        changeBtn.addEventListener("click", function(){
```

```
            const contentDiv =
document.getElementById("content");
            contentDiv.innerHTML = "<h2>New
Content</h2><p>Paragraph has been changed!</p>";
        });
    </script>
</body>
</html>
```

Using `textContent`

`textContent` gets or sets the text content of an element, stripping away any HTML.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Changing Content with textContent</title>
</head>
<body>
    <div id="content">
        <p>Original Paragraph</p>
    </div>
    <button id="changeBtn">Change Text</button>
    <script>
        const changeBtn = document.getElementById("changeBtn");
        changeBtn.addEventListener("click", function(){
            const contentDiv =
document.getElementById("content");
            contentDiv.textContent = "Text content has been
changed!";
        });
    </script>
</body>
```

```
</html>
```

Note: `textContent` is safer when inserting user-generated content as it prevents potential HTML injection.

Changing Styles

You can directly manipulate the style of elements using the `style` property.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Changing Styles</title>
</head>
<body>
    <p id="text">Change my color and font size!</p>
    <button id="styleBtn">Change Style</button>
    <script>
        const styleBtn = document.getElementById("styleBtn");
        styleBtn.addEventListener("click", function(){
            const text = document.getElementById("text");
            text.style.color = "blue";
            text.style.fontSize = "24px";
        });
    </script>
</body>
</html>
```

Adding and Removing Classes

Manipulating CSS classes is a cleaner way to change styles, allowing you to toggle predefined styles rather than setting individual properties.

Using `classList`

The `classList` property provides methods to add, remove, toggle, and check for classes.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Adding and Removing Classes</title>
    <style>
        .highlight {
            background-color: yellow;
            font-weight: bold;
        }
    </style>
</head>
<body>
    <p id="text">Click the button to highlight me!</p>
    <button id="toggleBtn">Toggle Highlight</button>
    <script>
        const toggleBtn = document.getElementById("toggleBtn");
        toggleBtn.addEventListener("click", function(){
            const text = document.getElementById("text");
            text.classList.toggle("highlight");
        });
    </script>
</body>
</html>
```

Methods:

- **`.add("className")`**: Adds a class.
- **`.remove("className")`**: Removes a class.
- **`.toggle("className")`**: Toggles a class on or off.
- **`.contains("className")`**: Checks if an element has a specific class.

Creating and Appending Elements

You can create new elements and add them to the DOM dynamically.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Creating and Appending Elements</title>
</head>
<body>
    <div id="container">
        <p>Existing Paragraph</p>
    </div>
    <button id="addBtn">Add New Paragraph</button>
    <script>
        const addBtn = document.getElementById("addBtn");
        addBtn.addEventListener("click", function(){
            const newPara = document.createElement("p");
            newPara.textContent = "This is a newly added
paragraph.";

document.getElementById("container").appendChild(newPara);
        });
    </script>
</body>
</html>
```

Explanation:

1. **`document.createElement("p")`**: Creates a new <p> element.
2. **`newPara.textContent`**: Sets the text content of the new paragraph.
3. **`appendChild(newPara)`**: Appends the new paragraph to the container div.

Additional Methods:

- **`insertBefore(newNode, referenceNode)`**: Inserts newNode before `referenceNode`.
- **`replaceChild(newNode, oldNode)`**: Replaces `oldNode` with newNode.
- **`removeChild(childNode)`**: Removes `childNode` from the DOM.

4. Event Handling

Events are actions that occur in the browser, such as clicks, form submissions, or key presses. Handling events allows you to make your webpage interactive.

Adding Event Listeners

Use `addEventListener` to attach event handlers to elements.

Syntax:

```
element.addEventListener("event", function);
```

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Event Handling</title>
</head>
<body>
    <button id="alertBtn">Show Alert</button>
    <script>
        const alertBtn = document.getElementById("alertBtn");
        alertBtn.addEventListener("click", function(){
            alert("Button was clicked!");
        });
    </script>
</body>
</html>
```

Removing Event Listeners

To remove an event listener, you must reference the exact function used when adding it.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Removing Event Listeners</title>
</head>
<body>
    <button id="removeBtn">Click Me</button>
    <button id="unregisterBtn">Unregister Click</button>
    <script>
        const removeBtn = document.getElementById("removeBtn");
        const unregisterBtn =
document.getElementById("unregisterBtn");
        function handleClick(){
            alert("Button clicked!");
        }
        removeBtn.addEventListener("click", handleClick);
        unregisterBtn.addEventListener("click", function(){
            removeBtn.removeEventListener("click", handleClick);
            alert("Click event removed.");
        });
    </script>
</body>
</html>
```

Explanation:

1. **`handleClick` Function:** Defined separately to allow removal.
2. **`addEventListener("click", handleClick)`**: Attaches the event.

3. **`removeEventListener("click", handleClick)`**: Removes the event when the "Unregister Click" button is pressed.

Event Object

When an event occurs, an event object is passed to the event handler containing information about the event.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Event Object</title>
</head>
<body>
    <input type="text" id="inputField" placeholder="Type
something...">
    <script>
        const inputField =
document.getElementById("inputField");
        inputField.addEventListener("keydown", function(event){
            console.log(`Key pressed: ${event.key}`);
        });
    </script>
</body>
</html>
```

Explanation:

- **`event.key`**: Logs the key that was pressed.

Common Properties:

- **`event.type`**: Type of the event (e.g., "click", "keydown").
- **`event.target`**: The element that triggered the event.

- **`event.preventDefault()`**: Prevents the default action associated with the event.

Common Events

- **`click`**: When an element is clicked.
- **`mouseover`**: When the mouse pointer is over an element.
- **`mouseout`**: When the mouse pointer leaves an element.
- **`keydown`**: When a key is pressed down.
- **`keyup`**: When a key is released.
- **`submit`**: When a form is submitted.
- **`change`**: When the value of an input changes.

Example: Handling Multiple Events

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Handling Multiple Events</title>
    <style>
        #box {
            width: 200px;
            height: 200px;
            background-color: #3498db;
            margin: 50px auto;
            transition: background-color 0.3s;
        }
    </style>
</head>
<body>
    <div id="box"></div>
    <script>
        const box = document.getElementById("box");
        box.addEventListener("mouseover", function(){
            box.style.backgroundColor = "#e74c3c";
```

```
        });
        box.addEventListener("mouseout", function(){
            box.style.backgroundColor = "#3498db";
        });
        box.addEventListener("click", function(){
            alert("Box clicked!");
        });
    </script>
</body>
</html>
```

Explanation:

- **`mouseover` and `mouseout` Events:** Change the box's color when hovered.
- **`click` Event:** Displays an alert when the box is clicked.

5. Traversing the DOM

Traversing the DOM involves moving through the document tree to access related elements.

Parent and Child Nodes

- **`parentNode`**: Accesses the parent of a selected node.
- **`children`**: Returns a live HTMLCollection of child elements.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Parent and Child Nodes</title>
</head>
<body>
    <div id="parentDiv">
        <p id="childP">This is a child paragraph.</p>
```

```
    </div>
    <script>
        const childP = document.getElementById("childP");
        const parentDiv = childP.parentNode;
        console.log(parentDiv); // Logs the <div id="parentDiv">
        const children = parentDiv.children;
        console.log(children); // Logs HTMLCollection containing
<p id="childP">
    </script>
</body>
</html>
```

Sibling Nodes

- **`nextSibling` and `previousSibling`**: Access the next and previous sibling nodes (including text nodes).
- **`nextElementSibling` and `previousElementSibling`**: Access the next and previous sibling elements, ignoring text nodes.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Sibling Nodes</title>
</head>
<body>
    <h1>Heading</h1>
    <p>First Paragraph</p>
    <p>Second Paragraph</p>
    <script>
        const firstPara = document.querySelector("p");
        const nextPara = firstPara.nextElementSibling;
        console.log(nextPara); // Logs the second <p> element
```

```
        const previousHeading =
firstPara.previousElementSibling;
        console.log(previousHeading); // Logs the <h1> element
    </script>
</body>
</html>
```

Accessing Elements

You can navigate the DOM tree to access various related elements.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Accessing Elements</title>
</head>
<body>
    <ul id="list">
        <li>Item 1</li>
        <li>Item 2
            <ul>
                <li>Subitem 1</li>
                <li>Subitem 2</li>
            </ul>
        </li>
        <li>Item 3</li>
    </ul>
    <script>
        const list = document.getElementById("list");
        const firstItem = list.firstElementChild;
        console.log(firstItem); // Logs <li>Item 1</li>
        const lastItem = list.lastElementChild;
        console.log(lastItem); // Logs <li>Item 3</li>
```

```
        const nestedList = list.children[1].querySelector("ul");
        console.log(nestedList); // Logs the nested <ul> with
subitems
    </script>
</body>
</html>
```

Explanation:

- **`firstElementChild` and `lastElementChild`**: Access the first and last child elements.
- **`children` Array-like Object**: Access child elements by index.
- **`querySelector` on Nested Elements**: Access deeper elements within a selected parent.

6. Modifying the DOM

Modifying the DOM involves adding, removing, replacing, or cloning elements to dynamically change the webpage's structure and content.

Adding Elements

Example: Adding a New List Item

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Adding Elements</title>
</head>
<body>
    <ul id="myList">
        <li>Existing Item</li>
    </ul>
    <button id="addItemBtn">Add Item</button>
    <script>
```

```
        const addItemBtn =
document.getElementById("addItemBtn");
        addItemBtn.addEventListener("click", function(){
            const newItem = document.createElement("li");
            newItem.textContent = "Newly Added Item";

document.getElementById("myList").appendChild(newItem);
        });
    </script>
</body>
</html>
```

Removing Elements

Example: Removing a List Item

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Removing Elements</title>
</head>
<body>
    <ul id="myList">
        <li>Item to Keep</li>
        <li id="removeMe">Item to Remove</li>
    </ul>
    <button id="removeBtn">Remove Item</button>
    <script>
        const removeBtn = document.getElementById("removeBtn");
        removeBtn.addEventListener("click", function(){
            const item = document.getElementById("removeMe");
            item.parentNode.removeChild(item);
            // Alternatively: item.remove();
        });
```

```
    </script>
</body>
</html>
```

Note: The `remove()` method is supported in modern browsers. For broader compatibility, use `parentNode.removeChild()`.

Replacing Elements

Example: Replacing a Paragraph with a Heading

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Replacing Elements</title>
</head>
<body>
    <p id="replaceMe">This paragraph will be replaced.</p>
    <button id="replaceBtn">Replace Paragraph</button>
    <script>
        const replaceBtn =
document.getElementById("replaceBtn");
        replaceBtn.addEventListener("click", function(){
            const oldElement =
document.getElementById("replaceMe");
            const newElement = document.createElement("h2");
            newElement.textContent = "This is the new Heading";
            oldElement.parentNode.replaceChild(newElement,
oldElement);
        });
    </script>
</body>
</html>
```

Cloning Elements

Example: Cloning a Div

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Cloning Elements</title>
    <style>
        .box {
            width: 100px;
            height: 100px;
            background-color: #8e44ad;
            margin: 10px;
            display: inline-block;
        }
    </style>
</head>
<body>
    <div class="box" id="originalBox"></div>
    <button id="cloneBtn">Clone Box</button>
    <script>
        const cloneBtn = document.getElementById("cloneBtn");
        cloneBtn.addEventListener("click", function(){
            const original =
document.getElementById("originalBox");
            const clone = original.cloneNode(true); // true for
deep clone
            clone.style.backgroundColor = "#e74c3c"; // Change
color to differentiate
            document.body.appendChild(clone);
        });
    </script>
</body>
</html>
```

Explanation:

- **`cloneNode(true)`**: Creates a deep clone, copying the element and all its descendants.
- **Styling Clones:** Adjust styles to differentiate cloned elements.

7. Forms and the DOM

Forms are essential for user input. Manipulating form elements via the DOM allows for dynamic interactions and validations.

Accessing Form Elements

Example: Accessing Input Fields

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Accessing Form Elements</title>
</head>
<body>
    <form id="myForm">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username">
        <label for="age">Age:</label>
        <input type="number" id="age" name="age">
        <button type="submit">Submit</button>
    </form>
    <script>
        const form = document.getElementById("myForm");
        form.addEventListener("submit", function(event){
            event.preventDefault();
            const username =
document.getElementById("username").value;
            const age = document.getElementById("age").value;
```

```
            console.log(`Username: ${username}, Age: ${age}`);
        });
    </script>
</body>
</html>
```

Reading and Setting Values

Example: Setting Input Values

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Setting Input Values</title>
</head>
<body>
    <input type="text" id="myInput" placeholder="Enter text">
    <button id="setValueBtn">Set Value</button>
    <script>
        const setValueBtn =
document.getElementById("setValueBtn");
        setValueBtn.addEventListener("click", function(){
            document.getElementById("myInput").value = "Hello,
World!";
        });
    </script>
</body>
</html>
```

Form Validation

Validating form data ensures that users provide the required and correctly formatted information.

Example: Simple Form Validation

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Form Validation</title>
    <style>
        .error {
            color: red;
            font-size: 0.9em;
        }
        .success {
            border-color: green;
        }
        .error-border {
            border-color: red;
        }
    </style>
</head>
<body>
    <form id="signupForm">
        <div>
            <label for="email">Email:</label>
            <input type="email" id="email" name="email">
            <span class="error" id="emailError"></span>
        </div>
        <div>
            <label for="password">Password (min 6
characters):</label>
            <input type="password" id="password"
name="password">
            <span class="error" id="passwordError"></span>
        </div>
        <button type="submit">Sign Up</button>
    </form>
```

```
    <script>
        const form = document.getElementById("signupForm");
        const emailInput = document.getElementById("email");
        const passwordInput =
document.getElementById("password");
        const emailError =
document.getElementById("emailError");
        const passwordError =
document.getElementById("passwordError");
        form.addEventListener("submit", function(event){
            event.preventDefault();
            let valid = true;
            // Validate Email
            if(emailInput.value.trim() === ""){
                emailError.textContent = "Email is required.";
                emailInput.classList.add("error-border");
                valid = false;
            } else {
                emailError.textContent = "";
                emailInput.classList.remove("error-border");
                emailInput.classList.add("success");
            }
            // Validate Password
            if(passwordInput.value.length < 6){
                passwordError.textContent = "Password must be at
least 6 characters.";
                passwordInput.classList.add("error-border");
                valid = false;
            } else {
                passwordError.textContent = "";
                passwordInput.classList.remove("error-border");
                passwordInput.classList.add("success");
            }
            if(valid){
```

```
                alert("Form submitted successfully!");
                // Proceed with form submission or further
processing
            }
        });
    </script>
</body>
</html>
```

Explanation:

- **Event Listener:** Handles form submission.
- **Validation Logic:** Checks if the email field is empty and if the password meets the length requirement.
- **Feedback:** Displays error messages and styles input fields based on validation.

Best Practices:

- **Use HTML5 Validation:** Utilize built-in validation attributes like `required`, `minlength`, and `type`.
- **Provide Clear Feedback:** Inform users about validation errors clearly and promptly.
- **Sanitize Inputs:** Always sanitize and validate data on the server side as well.

8. Advanced Topics

Delving deeper into DOM manipulation, these advanced topics will help you create more efficient and sophisticated web applications.

Event Delegation

Event delegation involves adding a single event listener to a parent element to manage events for its child elements, especially useful for dynamically added elements.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <title>Event Delegation</title>
    <style>
        ul {
            list-style-type: none;
        }
        li {
            padding: 8px;
            background-color: #ecf0f1;
            margin-bottom: 4px;
            cursor: pointer;
        }
        li:hover {
            background-color: #bdc3c7;
        }
    </style>
</head>
<body>
    <ul id="itemList">
        <li>Item 1</li>
        <li>Item 2</li>
    </ul>
    <button id="addItemBtn">Add Item</button>
    <script>
        const itemList = document.getElementById("itemList");
        const addItemBtn =
document.getElementById("addItemBtn");
        let itemCount = 3;
        // Event delegation: handle clicks on <li> elements
        itemList.addEventListener("click", function(event){
            if(event.target && event.target.nodeName === "LI"){
                alert(`${event.target.textContent} clicked!`);
            }
        });
```

```
        // Add new items dynamically
        addItemBtn.addEventListener("click", function(){
            const newItem = document.createElement("li");
            newItem.textContent = `Item ${itemCount}`;
            itemList.appendChild(newItem);
            itemCount++;
        });
    </script>
</body>
</html>
```

Explanation:

- **Single Event Listener:** Attached to the parent `<ul>` to handle all `<li>` clicks.
- **Dynamic Elements:** Newly added `<li>` elements inherit the click event handler without needing separate listeners.

Manipulating CSS Classes

Managing CSS classes is a powerful way to apply or remove styles based on interactions or states.

Example: Toggle Dark Mode

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Toggle Dark Mode</title>
    <style>
        body {
            background-color: white;
            color: black;
            transition: background-color 0.5s, color 0.5s;
        }
        .dark-mode {
            background-color: #2c3e50;
```

```
            color: #ecf0f1;
        }
        button {
            padding: 10px 20px;
            margin: 20px;
        }
    </style>
</head>
<body>
    <button id="toggleDarkBtn">Toggle Dark Mode</button>
    <p>This is some sample text.</p>
    <script>
        const toggleDarkBtn =
document.getElementById("toggleDarkBtn");
        toggleDarkBtn.addEventListener("click", function(){
            document.body.classList.toggle("dark-mode");
        });
    </script>
</body>
</html>
```

Explanation:

- **`.dark-mode` Class:** Defines dark mode styles.
- **`classList.toggle("dark-mode")`**: Adds or removes the class, effectively toggling dark mode.

Handling Multiple Elements

When dealing with multiple elements, it's essential to iterate over them efficiently.

Example: Highlight All Paragraphs on Click

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <title>Handling Multiple Elements</title>
    <style>
        .highlight {
            background-color: yellow;
        }
    </style>
</head>
<body>
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>
    <p>Paragraph 3</p>
    <button id="highlightBtn">Highlight All</button>
    <script>
        const highlightBtn =
document.getElementById("highlightBtn");
        highlightBtn.addEventListener("click", function(){
            const paragraphs = document.querySelectorAll("p");
            paragraphs.forEach(function(p){
                p.classList.toggle("highlight");
            });
        });
    </script>
</body>
</html>
```

Explanation:

- **`querySelectorAll("p")`**: Selects all <p> elements.
- **`.forEach()`**: Iterates over each paragraph to toggle the `highlight` class.

Performance Considerations

Efficient DOM manipulation can significantly enhance your webpage's performance.

Minimize Reflows and Repaints: Batch DOM changes to reduce the number of reflows and repaints.

```
const fragment = document.createDocumentFragment();
for(let i = 0; i < 100; i++){
    const newDiv = document.createElement("div");
    newDiv.textContent = `Div ${i}`;
    fragment.appendChild(newDiv);
}
document.body.appendChild(fragment);
```

Cache Selectors: Store references to frequently accessed elements.

```
const header = document.getElementById("header");
header.style.color = "blue";
header.style.backgroundColor = "lightgray";
```

Avoid Inline Styles: Use CSS classes for styling to leverage CSS optimizations.

```
// Instead of:
element.style.display = "none";
// Use:
element.classList.add("hidden");

.hidden {
    display: none;
}
```

- **Use Efficient Selectors:** Prefer ID and class selectors over tag selectors for faster element access.

9. Projects and Exercises

Applying what you've learned through projects and exercises is essential for mastering JavaScript DOM manipulation. Below are several hands-on activities to reinforce your understanding.

Exercise 1: Interactive To-Do List

Objective: Create a simple to-do list application where users can add, remove, and mark tasks as completed.

Features:

- Add new tasks.
- Remove existing tasks.
- Mark tasks as completed by clicking on them.

Solution Outline:

1. **HTML Structure:**
    - Input field for new tasks.
    - "Add Task" button.
    - Unordered list to display tasks.
2. **CSS Styling:**
    - Style the input field, button, and list.
    - Differentiate completed tasks with a line-through effect.
3. **JavaScript Functionality:**
    - Handle adding new tasks.
    - Handle removing tasks.
    - Toggle completion status on task click.

Sample Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Interactive To-Do List</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            max-width: 400px;
            margin: 50px auto;
        }
        input[type="text"] {
            width: 70%;
            padding: 10px;
            box-sizing: border-box;
        }
```

```
        button {
            padding: 10px;
            margin-left: 5px;
        }
        ul {
            list-style-type: none;
            padding-left: 0;
        }
        li {
            padding: 10px;
            background-color: #ecf0f1;
            margin-bottom: 5px;
            display: flex;
            justify-content: space-between;
            align-items: center;
            cursor: pointer;
        }
        li.completed {
            text-decoration: line-through;
            background-color: #bdc3c7;
        }
        .remove-btn {
            background-color: #e74c3c;
            border: none;
            color: white;
            padding: 5px 8px;
            cursor: pointer;
        }
    </style>
</head>
<body>
    <h2>To-Do List</h2>
    <input type="text" id="taskInput" placeholder="Enter a new
task">
```

```
    <button id="addTaskBtn">Add Task</button>
    <ul id="taskList"></ul>
    <script>
        const addTaskBtn =
document.getElementById("addTaskBtn");
        const taskInput = document.getElementById("taskInput");
        const taskList = document.getElementById("taskList");
        addTaskBtn.addEventListener("click", function(){
            const taskText = taskInput.value.trim();
            if(taskText !== ""){
                const li = document.createElement("li");
                li.textContent = taskText;
                const removeBtn =
document.createElement("button");
                removeBtn.textContent = "Remove";
                removeBtn.classList.add("remove-btn");
                li.appendChild(removeBtn);
                taskList.appendChild(li);
                taskInput.value = "";
            }
        });
        // Event delegation for removing and completing tasks
        taskList.addEventListener("click", function(event){
            const target = event.target;
            if(target.classList.contains("remove-btn")){
                const li = target.parentElement;
                taskList.removeChild(li);
            } else {
                target.classList.toggle("completed");
            }
        });
    </script>
</body>
</html>
```

Explanation:

- **Adding Tasks:** User enters a task and clicks "Add Task," which appends a new `<li>` with a remove button to the list.
- **Removing Tasks:** Clicking the "Remove" button deletes the corresponding task.
- **Completing Tasks:** Clicking on a task toggles the `completed` class, applying a line-through style.

Exercise 2: Image Gallery

Objective: Build an image gallery where clicking on a thumbnail displays the full-sized image.

Features:

- Display thumbnails of images.
- Clicking a thumbnail shows the full-sized image in a designated area.
- Optionally, include navigation to move between images.

Solution Outline:

1. **HTML Structure:**
    - Container for thumbnails.
    - Display area for the full-sized image.
2. **CSS Styling:**
    - Style thumbnails and the display area.
    - Add hover effects to thumbnails.
3. **JavaScript Functionality:**
    - Handle thumbnail clicks to update the display area.
    - Optionally, add navigation buttons (Next, Previous).

Sample Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Image Gallery</title>
    <style>
        body {
```

```
            font-family: Arial, sans-serif;
            max-width: 800px;
            margin: 50px auto;
            text-align: center;
        }
        #gallery {
            display: flex;
            justify-content: center;
            flex-wrap: wrap;
            margin-bottom: 20px;
        }
        .thumbnail {
            width: 100px;
            height: 100px;
            margin: 5px;
            object-fit: cover;
            cursor: pointer;
            border: 2px solid transparent;
            transition: border 0.3s;
        }
        .thumbnail:hover {
            border: 2px solid #3498db;
        }
        #fullImage {
            width: 100%;
            max-height: 500px;
            object-fit: contain;
            border: 1px solid #bdc3c7;
        }
    </style>
</head>
<body>
    <h2>Image Gallery</h2>
    <div id="gallery">
```

```
        <img src="images/thumb1.jpg" alt="Image 1"
class="thumbnail" data-full="images/full1.jpg">
        <img src="images/thumb2.jpg" alt="Image 2"
class="thumbnail" data-full="images/full2.jpg">
        <img src="images/thumb3.jpg" alt="Image 3"
class="thumbnail" data-full="images/full3.jpg">
        <!-- Add more thumbnails as needed -->
    </div>
    <img id="fullImage" src="images/full1.jpg" alt="Full Size
Image">
    <script>
        const thumbnails =
document.querySelectorAll(".thumbnail");
        const fullImage = document.getElementById("fullImage");
        thumbnails.forEach(function(thumbnail){
            thumbnail.addEventListener("click", function(){
                const fullSrc = this.getAttribute("data-full");
                fullImage.src = fullSrc;
                fullImage.alt = this.alt;
            });
        });
    </script>
</body>
</html>
```

Explanation:

- **Thumbnails:** Small versions of images with a `data-full` attribute pointing to the full-sized image.
- **Display Area:** Shows the full-sized image, initially set to the first image.
- **Event Listeners:** Clicking a thumbnail updates the `src` and `alt` of the `#fullImage`.

Exercise 3: Form Validation

Objective: Enhance form validation by providing real-time feedback as users fill out the form.

Features:

- Real-time validation for input fields.
- Display validation messages.
- Highlight valid and invalid fields.

Solution Outline:

1. **HTML Structure:**
    - Form with various input fields (e.g., name, email, password).
2. **CSS Styling:**
    - Style the form and validation messages.
    - Indicate valid and invalid fields with colors or borders.
3. **JavaScript Functionality:**
    - Validate inputs on input events.
    - Display or hide validation messages accordingly.
    - Prevent form submission if validation fails.

Sample Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Real-Time Form Validation</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            max-width: 500px;
            margin: 50px auto;
        }
        form div {
            margin-bottom: 15px;
            position: relative;
        }
```

```
    label {
        display: block;
        margin-bottom: 5px;
    }
    input {
        width: 100%;
        padding: 8px;
        box-sizing: border-box;
    }
    .error-message {
        color: red;
        font-size: 0.9em;
        position: absolute;
        top: 100%;
        left: 0;
    }
    .valid {
        border: 2px solid green;
    }
    .invalid {
        border: 2px solid red;
    }
    button {
        padding: 10px 20px;
        background-color: #2ecc71;
        border: none;
        color: white;
        cursor: pointer;
    }
    button:disabled {
        background-color: #95a5a6;
        cursor: not-allowed;
    }
</style>
```

```
</head>
<body>
    <h2>Register</h2>
    <form id="registrationForm">
        <div>
            <label for="username">Username:</label>
            <input type="text" id="username" name="username">
            <span class="error-message"
id="usernameError"></span>
        </div>
        <div>
            <label for="email">Email:</label>
            <input type="email" id="email" name="email">
            <span class="error-message" id="emailError"></span>
        </div>
        <div>
            <label for="password">Password (min 6
characters):</label>
            <input type="password" id="password"
name="password">
            <span class="error-message"
id="passwordError"></span>
        </div>
        <button type="submit" id="submitBtn"
disabled>Submit</button>
    </form>
    <script>
        const registrationForm =
document.getElementById("registrationForm");
        const usernameInput =
document.getElementById("username");
        const emailInput = document.getElementById("email");
        const passwordInput =
document.getElementById("password");
```

```
        const submitBtn = document.getElementById("submitBtn");
        const usernameError =
document.getElementById("usernameError");
        const emailError =
document.getElementById("emailError");
        const passwordError =
document.getElementById("passwordError");
        let isUsernameValid = false;
        let isEmailValid = false;
        let isPasswordValid = false;
        // Validate Username
        usernameInput.addEventListener("input", function(){
            const username = usernameInput.value.trim();
            if(username.length >= 3){
                usernameError.textContent = "";
                usernameInput.classList.remove("invalid");
                usernameInput.classList.add("valid");
                isUsernameValid = true;
            } else {
                usernameError.textContent = "Username must be at
least 3 characters.";
                usernameInput.classList.remove("valid");
                usernameInput.classList.add("invalid");
                isUsernameValid = false;
            }
            toggleSubmit();
        });
        // Validate Email
        emailInput.addEventListener("input", function(){
            const email = emailInput.value.trim();
            const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
            if(emailRegex.test(email)){
                emailError.textContent = "";
                emailInput.classList.remove("invalid");
```

```
                emailInput.classList.add("valid");
                isEmailValid = true;
            } else {
                emailError.textContent = "Please enter a valid
email.";
                emailInput.classList.remove("valid");
                emailInput.classList.add("invalid");
                isEmailValid = false;
            }
            toggleSubmit();
        });
        // Validate Password
        passwordInput.addEventListener("input", function(){
            const password = passwordInput.value;
            if(password.length >= 6){
                passwordError.textContent = "";
                passwordInput.classList.remove("invalid");
                passwordInput.classList.add("valid");
                isPasswordValid = true;
            } else {
                passwordError.textContent = "Password must be at
least 6 characters.";
                passwordInput.classList.remove("valid");
                passwordInput.classList.add("invalid");
                isPasswordValid = false;
            }
            toggleSubmit();
        });
        // Toggle Submit Button
        function toggleSubmit(){
            if(isUsernameValid && isEmailValid &&
isPasswordValid){
                submitBtn.disabled = false;
            } else {
```

```
                submitBtn.disabled = true;
            }
        }
        // Handle Form Submission
        registrationForm.addEventListener("submit",
function(event){
            event.preventDefault();
            alert("Form submitted successfully!");
            // Further processing can be done here
        });
    </script>
</body>
</html>
```

Explanation:

- **Real-Time Validation:** Each input field is validated as the user types.
- **Visual Feedback:** Valid inputs are highlighted in green, invalid in red.
- **Error Messages:** Displayed below each input field.
- **Submit Button:** Enabled only when all fields are valid.

Exercise 4: Dynamic Content Loading

Objective: Implement a "Load More" feature that fetches additional content from an external API and appends it to the webpage.

Features:

- Initial set of content displayed.
- "Load More" button to fetch and display more content.
- Handle cases when no more content is available.

Solution Outline:

1. **HTML Structure:**
    - Container for initial content.
    - "Load More" button.
2. **CSS Styling:**
    - Style the content container and button.

3. **JavaScript Functionality:**
    - Fetch data from an API (e.g., JSONPlaceholder).
    - Append new content to the container.
    - Disable the button when no more data is available.

Sample Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Dynamic Content Loading</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            max-width: 800px;
            margin: 50px auto;
            text-align: center;
        }
        .post {
            border: 1px solid #bdc3c7;
            padding: 15px;
            margin-bottom: 15px;
            text-align: left;
        }
        #loadMoreBtn {
            padding: 10px 20px;
            background-color: #2980b9;
            color: white;
            border: none;
            cursor: pointer;
        }
        #loadMoreBtn:disabled {
            background-color: #95a5a6;
            cursor: not-allowed;
        }
```

```
    </style>
</head>
<body>
    <h2>Dynamic Content Loading</h2>
    <div id="postsContainer">
        <!-- Initial posts will be loaded here -->
    </div>
    <button id="loadMoreBtn">Load More</button>
    <script>
        const postsContainer =
document.getElementById("postsContainer");
        const loadMoreBtn =
document.getElementById("loadMoreBtn");
        let currentPage = 1;
        const limit = 5; // Number of posts per page
        const totalPages = 4; // Total pages available (for
demonstration)
        // Function to fetch posts
        async function fetchPosts(page, limit){
            try{
                const response = await
fetch(`https://jsonplaceholder.typicode.com/posts?_page=${page}&
_limit=${limit}`);
                const data = await response.json();
                return data;
            } catch(error){
                console.error("Error fetching posts:", error);
                return [];
            }
        }
        // Function to display posts
        function displayPosts(posts){
            posts.forEach(post => {
                const postDiv = document.createElement("div");
```

```
            postDiv.classList.add("post");
            postDiv.innerHTML = `
                <h3>${post.title}</h3>
                <p>${post.body}</p>
            `;
            postsContainer.appendChild(postDiv);
        });
    }
    // Initial Load
    async function initialLoad(){
        const posts = await fetchPosts(currentPage, limit);
        displayPosts(posts);
    }
    initialLoad();
    // Load More Button Click
    loadMoreBtn.addEventListener("click", async function(){
        currentPage++;
        const posts = await fetchPosts(currentPage, limit);
        if(posts.length > 0){
            displayPosts(posts);
            if(currentPage >= totalPages){
                loadMoreBtn.disabled = true;
                loadMoreBtn.textContent = "No More Posts";
            }
        } else {
            loadMoreBtn.disabled = true;
            loadMoreBtn.textContent = "No More Posts";
        }
    });
</script>
</body>
</html>
```

Explanation:

- **Initial Load:** Fetches the first set of posts and displays them.
- **Load More:** Fetches subsequent pages upon clicking the button.
- **Termination:** Disables the "Load More" button when all pages are loaded.

Note: The `totalPages` variable is set for demonstration. In a real-world scenario, you'd determine if more data is available based on the API's response headers or data.

Exercise 5: Modal Popup

Objective: Create a modal popup that appears when a button is clicked and can be closed by clicking a close button or outside the modal.

Features:

- Open modal on button click.
- Close modal by clicking the close button or outside the modal content.
- Prevent background scrolling when the modal is open.

Solution Outline:

1. **HTML Structure:**
    - Button to open the modal.
    - Modal container with content and a close button.
2. **CSS Styling:**
    - Style the modal overlay and content.
    - Hide the modal by default.
3. **JavaScript Functionality:**
    - Handle opening and closing the modal.
    - Close the modal when clicking outside the content area.

Sample Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Modal Popup</title>
    <style>
        body.modal-open {
            overflow: hidden;
```

```
        }
        /* Modal Overlay */
        .modal {
            display: none; /* Hidden by default */
            position: fixed; /* Stay in place */
            z-index: 1000; /* Sit on top */
            left: 0;
            top: 0;
            width: 100%; /* Full width */
            height: 100%; /* Full height */
            overflow: auto; /* Enable scroll if needed */
            background-color: rgba(0,0,0,0.5); /* Black w/
opacity */
        }
        /* Modal Content */
        .modal-content {
            background-color: #fefefe;
            margin: 15% auto; /* 15% from top and centered */
            padding: 20px;
            border: 1px solid #888;
            width: 80%; /* Could be more or less, depending on
screen size */
            max-width: 500px;
            position: relative;
            border-radius: 5px;
        }
        /* Close Button */
        .close-btn {
            color: #aaa;
            position: absolute;
            top: 10px;
            right: 15px;
            font-size: 28px;
            font-weight: bold;
```

```
            cursor: pointer;
        }
        .close-btn:hover,
        .close-btn:focus {
            color: black;
            text-decoration: none;
        }
    </style>
</head>
<body>
    <h2>Modal Popup Example</h2>
    <button id="openModalBtn">Open Modal</button>
    <!-- The Modal -->
    <div id="myModal" class="modal">
        <!-- Modal content -->
        <div class="modal-content">
            <span class="close-btn">&times;</span>
            <h3>Modal Header</h3>
            <p>This is a sample modal popup.</p>
        </div>
    </div>
    <script>
        const openModalBtn =
document.getElementById("openModalBtn");
        const modal = document.getElementById("myModal");
        const closeBtn = document.querySelector(".close-btn");
        // Function to open the modal
        function openModal(){
            modal.style.display = "block";
            document.body.classList.add("modal-open");
        }
        // Function to close the modal
        function closeModal(){
            modal.style.display = "none";
```

```
            document.body.classList.remove("modal-open");
        }
        // Event listeners
        openModalBtn.addEventListener("click", openModal);
        closeBtn.addEventListener("click", closeModal);
        // Close modal when clicking outside the modal content
        window.addEventListener("click", function(event){
            if(event.target === modal){
                closeModal();
            }
        });
    </script>
</body>
</html>
```

Explanation:

- **HTML:** Defines a button to open the modal and the modal structure with a close button.
- **CSS:** Styles the modal overlay and content, ensuring it's centered and hidden by default.
- **JavaScript:**
    - **Opening the Modal:** Displays the modal and adds a class to prevent background scrolling.
    - **Closing the Modal:** Hides the modal and removes the class.
    - **Click Outside to Close:** Detects clicks outside the modal content to close it.

Enhancements:

- **Accessibility:** Add ARIA attributes and manage focus when the modal is open.
- **Animations:** Incorporate fade-in and fade-out effects for smoother transitions.

10. Multiple Choice Questions

Test your understanding of JavaScript DOM manipulation with the following multiple-choice questions. Answers and explanations are provided after each question.

Question 1

What does DOM stand for in web development?

A) Document Object Model
B) Data Object Method
C) Document Oriented Model
D) Data Object Model

Answer: A) Document Object Model

Explanation:

- **Document Object Model (DOM)** is a programming interface for HTML and XML documents.

Question 2

Which method is used to select an element by its ID?

A) `document.querySelector("#id")`
B) `document.getElementById("id")`
C) `document.querySelectorAll("#id")`
D) Both A and B

Answer: D) Both A and B

Explanation:

- **`document.getElementById("id")`** directly selects the element with the specified ID.
- **`document.querySelector("#id")`** also selects the element with the specified ID using a CSS selector.

Question 3

How can you change the text content of an element with the ID `myElement`?

A) `document.getElementById("myElement").innerHTML = "New Text";`
B) `document.getElementById("myElement").textContent = "New Text";`

C) `document.querySelector("#myElement").innerText = "New Text";`
D) All of the above

Answer: D) All of the above

Explanation:

- **`innerHTML`**, **`textContent`**, and **`innerText`** can all be used to change the text content, though they have subtle differences:
  - **`innerHTML`** parses content as HTML.
  - **`textContent`** sets or returns the text content without parsing HTML.
  - **`innerText`** is similar to `textContent` but considers CSS styling and layout.

Question 4

Which property is used to change the background color of an element?

A) `element.color`
B) `element.background`
C) `element.style.backgroundColor`
D) `element.style.color`

Answer: C) `element.style.backgroundColor`

Explanation:

- **`element.style.backgroundColor`** is the correct property to change the background color of an element.

Question 5

What is the purpose of the `addEventListener` method?

A) To add a new HTML element to the DOM
B) To listen for and handle events on an element
C) To remove an existing event listener
D) To prevent default browser behavior

Answer: B) To listen for and handle events on an element

Explanation:

- **`addEventListener`** attaches an event handler to an element to listen for specific events like clicks, hovers, etc.

Question 6

Which method would you use to create a new HTML element?

A) `document.createElement()`
B) `document.newElement()`
C) `document.appendElement()`
D) `document.addElement()`

Answer: A) `document.createElement()`

Explanation:

- **`document.createElement()`** creates a new HTML element of the specified type.

Question 7

How can you remove an element from the DOM?

A) `element.remove()`
B) `element.parentNode.removeChild(element)`
C) `element.delete()`
D) Both A and B

Answer: D) Both A and B

Explanation:

- **`element.remove()`** directly removes the element from the DOM.
- **`element.parentNode.removeChild(element)`** removes the element by accessing its parent.

Note: The `remove()` method is supported in modern browsers.

Question 8

Which property of the event object contains the element that triggered the event?

A) `event.currentTarget`
B) `event.target`
C) `event.source`
D) `event.origin`

Answer: B) `event.target`

Explanation:

- **`event.target`** refers to the actual element that triggered the event.

Question 9

What does the `textContent` property do?

A) Sets or returns the HTML content of an element
B) Sets or returns the text content of an element
C) Sets or returns the value of a form element
D) Removes all child nodes of an element

Answer: B) Sets or returns the text content of an element

Explanation:

- **`textContent`** retrieves or sets the text content of an element without parsing HTML.

Question 10

Which method would you use to add a new class to an element?

A) `element.className += " newClass"`
B) `element.classList.add("newClass")`
C) `element.setAttribute("class", "newClass")`
D) Both A and B

Answer: D) Both A and B

Explanation:

- **`element.className += " newClass"`** appends a new class to the existing classes.
- **`element.classList.add("newClass")`** adds a new class using the `classList` API, which is more robust.

Question 11

What will `document.querySelectorAll(".item")[0]` return?

A) The first element with the class `item`
B) All elements with the class `item`
C) An array of elements with the class `item`
D) Undefined

Answer: A) The first element with the class `item`

Explanation:

- **`document.querySelectorAll(".item")`** returns a NodeList of all elements with the class `item`.
- **`[0]`** accesses the first element in the NodeList.

Question 12

Which of the following is NOT a valid way to change the source of an image element?

A) `imgElement.src = "newImage.jpg";`
B) `imgElement.setAttribute("src", "newImage.jpg");`
C) `imgElement.replace("src", "newImage.jpg");`
D) Both A and B are valid

Answer: C) `imgElement.replace("src", "newImage.jpg");`

Explanation:

- **Option C** is invalid because `replace` is not a method for setting attributes.
- **Options A and B** are both valid ways to change an image's `src` attribute.

Question 13

How can you prevent a form from submitting and reloading the page?

A) `event.stopPropagation()`
B) `event.preventDefault()`
C) `return false;`
D) Both B and C

Answer: D) Both B and C

Explanation:

- **`event.preventDefault()`** prevents the default action (form submission).
- **`return false;`** inside an event handler also prevents default actions and stops propagation.

Question 14

What does the `cloneNode(true)` method do?

A) Clones the node without its children
B) Clones the node along with all its child nodes
C) Moves the node to a new location
D) Deletes the node after cloning

Answer: B) Clones the node along with all its child nodes

Explanation:

- **`cloneNode(true)`** creates a deep clone, copying the node and all its descendants.
- **`cloneNode(false)`** would clone the node without its children.

Question 15

Which of the following is used to access form data in JavaScript?

A) `document.forms`
B) `document.getElementById("formId").elements`

C) `new FormData(formElement)`
D) All of the above

Answer: D) All of the above

Explanation:

- **`document.forms`** accesses all forms in the document.
- **`document.getElementById("formId").elements`** accesses form controls within a specific form.
- **`new FormData(formElement)`** creates a `FormData` object for easy form data manipulation.

11. Conclusion

Congratulations! You've completed the comprehensive guide to **JavaScript DOM Manipulation**. This guide has covered everything from the basics of selecting and manipulating elements to more advanced topics like event delegation and performance optimization. By working through the code examples, exercises, and multiple-choice questions, you've built a solid foundation in DOM manipulation that will empower you to create dynamic and interactive web applications.